The Next 700
Domain Specific Languages

Nicolas Wu
Imperial College London
Peter Landin Seminar, BCS 2022
functional

> imperative
London, 1666
“When trying to get from England to Holland, I was detained for some time in the Thames by adverse winds. During that time, knowing not what to do, and having nobody in the ship except sailors, I meditated on things, and especially thought about my old design of a rational language [...]”

“For if we had such as I imagine, we could reason about metaphysics and morality just like we do about geometry and analysis [...]”

“...a proof of the existence of God from some hypotheses and axioms about motion
an alphabet of human thought, the “alphabetum cogitationum humanarium”
the idea that concepts are combinations of a core set of concepts

“...or if praise is given to the men who have determined the number of regular solids [...] how much better will it be to bring under mathematical laws human reasoning, which is the most excellent and useful thing we have.” (Davis, 2018)
"Wir müssen wissen. Wir werden wissen!"
We must know. We shall know!
Hilbert, 1930,
Congress of German Scientists and Physicians

"Ignoramus et ignorabimus"
We do not know and we shall not know
du Bois-Reynold, 1872,
Congress of German Scientists and Physicians

Hilbert and Ackerman (1928)
Formalised the Entscheidungsproblem,
the "decision problem":
Given an input statement in first order predicate calculus, is there an algorithm that answers either "yes" or "no" if the statement is provable?
Gödel (1931) showed that any reasonable system of logic has true statements that cannot be proven within that system.

but if the Entscheidungsproblem holds true, then perhaps an algorithm can prove statements?

Church (1936) and Turing (1936) Independently answered the Entscheidungsproblem: No! Such an algorithm is not possible.

A NOTE ON THE Entscheidungsproblem

ALONZO CHURCH

In a recent paper the author has proposed a definition of the commonly used term "effectively calculable" and has shown on this basis the decision that the general case of the Entscheidungsproblem is unsolvable in any system of symbolic logic which is adequate to a certain portion of arithmetic and is $\omega$-consistent. The purpose of the present note is to realize an extension of this result to the general function-algorithm of Hilbert and Ackermann.

In the author's chief paper it is proved that there can be associated recursively with every well-formed formula a recursive enumeration of the formals into which it is convertible. This means the existence of a recursively defined function $f$ of two positive integers such that, if $y$ is the Gödel representation of a well-formed formula $\phi$, then $f(x, y)$ is the Gödel representation of the $x$th formula in the enumeration of the formula into which $\phi$ is convertible.

Consider the system $L$ of symbolic logic whose atomic symbols are $\text{true}$, $\text{false}$, and whose function symbols are $\text{and}$, $\text{or}$, $\text{not}$, and whose predicate symbols are $\text{for all}$, $\text{there exists}$.

...
The Lambda Calculus

\(\lambda\)-calculus is about bindings and substitutions

Instead of

\[ f(x) = 5x + 3 \]

we can isolate \(f\):

\[ f = \lambda x. 5x + 3 \]

The definition of "effectively calculable"

Church numerals:

\[
\begin{align*}
1 &= \lambda s z . s z \\
2 &= \lambda s z . s (s z) \\
3 &= \lambda s z . s (s (s z)) \\
&\vdots \\
m + n &= \lambda s z . m s (n s z) \\
_+ &= \lambda m n s z . m s (n s z)
\end{align*}
\]

A proof that converting a term \(A\) into \(B\) is undecidable

1936

A SET OF POSTULATES FOR THE FOUNDATION OF LOGIC.

BY ALONZO CHURCH.

AN UNSOLVABLE PROBLEM OF ELEMENTARY NUMBER THEORY.

BY ALONZO CHURCH.

1. Introduction. There is a class of problems of elementary number theory which can be stated in the form that it is required to find an effectively calculable function \(f\) of positive integers, such that \(f(x, s_0, \ldots, s_m) = 0\) is a necessary and sufficient condition for the truth of a certain proposition of elementary number theory involving \(s_0, \ldots, s_m\) as free variables.

An example of such a problem is the problem of finding of any given positive integer \(a\) whether or not there exist positive integers \(x, y, z\) such that \(a^2 + b^2 = c^2\). For this may be interpreted, required to find an effectively calculable function \(f\) such that \(f(a)\) is equal to \(0\) if and only if there exist positive integers \(x, y, z\) such that \(a^2 + b^2 = c^2\). Clearly the condition that the function \(f\) be effectively calculable is an essential part of the problem, since without it the problem becomes trivial.

In his chief paper, it is pointed out that there can be found for every well-formed formula \(F\) a sentence \(\mathcal{S}_F\) in the language of the formal system \(\mathcal{L}\) such that

\[\mathcal{S}_F \iff F\]
Turing Machines

Automatic machines.

If at each stage the motion of a machine (in the sense of §1) is completely determined by the configuration, we shall call the machine an "automatic machine" (or \( e \)-machine).

Computing machines.

If an \( e \)-machine prints two kinds of symbols, of which the first kind (called figures) consists entirely of 0 and 1 (the others being called symbols of the second kind), then the machine will be called a computing machine.

We can show further that there can be no machine \( \mathcal{E} \) which, when supplied with the S.D of an arbitrary machine \( M \), will determine whether \( M \) ever prints a given symbol (0 say).

The results of §8 have some important applications. In particular, they can be used to show that the Hilbert Entscheidungsproblem can have no solution. For the present I shall confine myself to proving this particular solution.

Turing showed construction of a formula that takes in a Turing machine as input and is provable only if the machine halts but he also showed that no machine can determine if an arbitrary machine halts.
John von Neumann 1903 - 1957

Electronic Delay Storage Automatic Calculator, 1948

1945

Electronic Discrete Variable Automatic Computer, 1949

Dedicated to the
United States Army Ground Forces and
the
University of Pennsylvania

June 30, 1945
Early Programming Languages
Much of my work has come from being lazy. I didn’t like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs.

[... ] Most people think FORTRAN’s main contribution was to enable the programmer to write programs in algebraic formulas instead of machine language. But it isn’t. What FORTRAN did primarily was to mechanize the organization of loops.

John Backus in “Think, IBM, July/August 1979”
LISP, 1958

If \( \mathcal{E} \) is a form in variables \( x_1, \ldots, x_n \), then \( \lambda(x_1, \ldots, x_n) \mathcal{E} \) will be taken to be the function of \( x_1, \ldots, x_n \) in that order in \( \mathcal{E} \) and evaluating the resulting expression. For example, \( \lambda((x,y),y+x) \) is a function of two variables, and \( \lambda((x,y),y+x)(3,4) = 10 \).

Conditional expressions are a device for expressing the dependence of quantities on propositional quantities. A conditional expression has the form

\[
(p_1 \rightarrow e_1, \ldots, p_n \rightarrow e_n)
\]

where the \( p_i \)s are propositional expressions and the \( e_i \)s are expressions of any kind. It may be read, "If \( p_1 \) then \( e_1 \), otherwise if \( p_2 \) then \( e_2 \), otherwise if \( p_n \) then \( e_n \)" or "\( p_1 \) yields \( e_1 \), \ldots, \( p_n \) yields \( e_n \)"

By using conditional expressions we can, without circularly, define functions by formula in which the defined function occurs. For example, we write

\[
n! = (n! - 1, T \rightarrow n \rightarrow (n-1)!)\]

When we use this formula to evaluate \( 0! \) we get the answer 1;

G. Functions with Functions as Arguments

There are a number of useful functions some of whose arguments are functions. They are especially useful in defining other functions. One such function is \( \text{maplist}(x,f) \) with an expression \( x \) and an argument \( f \) that is a function from expression argument \( x \) and an argument \( f \) is a function from expression argument \( x \) and an argument \( f \).

\[
\text{maplist}(x,f) = [\text{null}(x) \rightarrow \text{NIL}; T \rightarrow \text{cons}(f(x), \text{maplist}(\text{cdr}(x), f))]
\]

Features of Lisp

- Based on lambda calculus
- Conditionals
- Recursive functions
- List processing (car, cdr, cons)
- Higher-order functions

Artificial intelligence Group

J. McCarthy
K. Brayton
P. Edwards
P.Fox
L. Sebea
D. Loomis
X. Maling
E. Park
S. Russell

An s-expression \( x \) that is not atomic is represented by a word, the address and decrement parts of which contain the locations of the subexpressions \( \text{car}(x) \) and \( \text{cdr}(x) \), respectively. In the list form of expressions the s-expression \( (A, (B, C), D) \) is represented by the list structure of Fig. 2.

When a list structure is regarded as representing a list, we see that each term of the list occupies the address part of a word, the decrement part of which points to the word containing the next term, while the last word has NIL in its decrement. The dot notation, e.g. \( (a.b) \), which is discussed in Section 0.2, is not allowed in LISP I; all lists and sublists must end with NIL.
... and Green, Katz, Perlis, Rutishauser, Lamelaon, van Wijngaarden, Vauguis, Wegstein, and Woodger

---

**Algol 1958**

- Iteration and recursion*
- Distinct assignment (:=) and equality (=)
- Code blocks and lexical scope
- Nested procedure, and procedure passing
- BNF
- Call-by-value and call-by-name

---

```
procedure Absmax(a:array[1..n] of integer; i,k:integer) Result: real
Subscripts: (i,k)
comment The absolute greatest element of the matrix a, of size n by m is transferred to y, and the subscripts of this element to i and k
array a ; integer n, m, i, k ; real y ;
begin integer p, q ;
y := 0 ;
for p := 1 step 1 until n do for q := 1 step 1 until m do
if abs(a[p,q]) > y then begin y := abs(a[p,q]) ; i := p ; k := q ;
end end Absmax
```

---

"... a language so far ahead of its time, that it was not only an improvement on its predecessors, but also on nearly all its successors" – Tony Hoare, 1974, in Hints on Programming Language Design

---

John Backus 1924 - 2007

John McCarthy 1927 - 2011

Peter Naur 1922 - 2016

---

Revised report on the algorithmic language Algol 60

Dedicated to the memory of William TAIT

by


Edited by Peter Naur

Approved by the council of the International Federation for Information Processing
Algol and Lambdas, 1965

Here is a typical form of AE, presented both informally and formally:

\[
\begin{align*}
&\text{let } a = A, \\
&\text{and } b = B, \\
&\text{and } f(x, y) = F, \\
&\text{let rec } g(z) = G, \\
&\text{and } h(y, z) = H, \\
&\text{let } k(u, v) = K.
\end{align*}
\]

The only consideration in choosing between let and where will be the relative convenience of writing an auxiliary definition before or after the expression it qualifies.

<table>
<thead>
<tr>
<th>TABLE 1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Identifiers, operator symbols, also some special words and configurations</th>
<th>Identifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local identifiers</td>
<td>Variables bound in a λ-expression occurring as operator</td>
</tr>
<tr>
<td>Formal parameters</td>
<td>Variables bound in a λ-expression occurring as operand</td>
</tr>
<tr>
<td>Function designator, subscripted variable, and procedure statement</td>
<td>Operator/operator combination</td>
</tr>
<tr>
<td>Procedure</td>
<td>( \lambda (\cdot) )-expression</td>
</tr>
<tr>
<td>Actual parameter called by name</td>
<td></td>
</tr>
</tbody>
</table>
ISWIM, 1966

Peter Landin
1930 - 2009

The Next 700 Programming Languages

A family of non-procedural computer languages is de-
defined in terms of a small set of common operations and
operators, each of which can be applied to any program or
expression. The family contains a language for each of
the three major computer languages: assembly, machine
languages, and high-level languages. This is done by
simply defining the necessary operations for each of
them, without providing language features

1. Introduction

Most programming languages are made up of a set of
operations, each of which has its own set of rules for
how it can be used. In ISWIM, the operations are
arranged into a hierarchy, where each operation can
be applied to any program or expression, without
requiring special syntax for each language. This
approach allows for a more flexible and powerful
language, where different operations can be
combined in various ways to create complex
expressions.

2. Mathematical notation for functions, using let
and where clauses

3. Indentation to separate clauses (offside rule)

4. A distinction between physical and logical
languages

5. Interpretation through layers of abstraction
linked by syntax, grammar, and denotation

6. Garbage collection

7. Equivalences between terms

8. No explicit sequencing

9. Algebraic datatypes
Church without Lambda

$$(u - 1)(u + 2)$$ \{\lambda u. (u - 1)(u + 2)\}[7 - 3]

where \(u = 7 - 3\).

$${u(u + 1) - v(v + 1)}\ \{\lambda (u, v). u(u + 1) - v(v + 1)\}$$

where \(u = 2p + q\) and \(v = p - 2q\).

$$f(7 - 3)$$

where rec \(f(n) = \begin{cases} \lambda f. f(7 - 3) & \text{if } n = 0 \text{ then } 1 \\ \text{else } nf(n - 1) \end{cases}$$

where \(n = \text{round}(n)\)

instead of the specification

Regarding indentation, in many ways I am in sympathy with this, but [...] you would regret it because of the kind of rearrangement of manuscripts done in printing

where notation

• More convenient that lambda terms
• Indentation sensitive
• Multiple and recursive definitions
• Used for types

The Next 700 Programming Languages

By P. J. Landin

Peter Naur 1928 - 2016

The mechanical evaluation of expressions

By P. J. Landin

This report is contributed to the “memory” of the ability of good computer science of man and machine is, as Dijkstra said, that it is a problem even more than the ability of solving the logical problem and the ability to work

[Peter Naur's quote about indentation]

Peter Naur 1928 - 2016
Language Abstraction

The “physical/logical” terminology is often used to distinguish features that are a fortuitous consequence of physical conditions from features that are in some sense more essential. This idea is carried further by making a similar distinction among the “more essential” features. In fact, ISWIM is presented here as a four-level concept comprising the following:

1. **Physical ISWIM**
   - **Tokenize**
   - **Parse**
   - **Interpret**
   - **Evaluate**

2. **Logical ISWIM**

3. **Abstract ISWIM**

4. **Applicative Expression**

5. **Value**

\[ f(b + 2c) \]
where \( f(x) = x(x+a) \)

1/10th of the paper is devoted to the idea of abstraction layers for defining a language!

Today this is regarded as the standard means of giving language semantics.
Equivalences

Enabling equational reasoning featured heavily in this paper

(1) subexpressions

\[ (\mu) \text{ If } L \equiv L' \text{ then } L(M) \equiv L'(M) \]

\[ (\nu) \text{ If } M \equiv M' \text{ then } L(M) \equiv L(M') \]

(2) definitions

\[ \text{(let) } \text{ let } x = M; \ L \equiv L \text{ where } x = M \]

(3) built-in expressions

\[ \text{true} \rightarrow M; \ N \equiv M \]

\[ \text{false} \rightarrow M; \ N \equiv N \]

(4) primitives

A problem-orientation of Iswim can be characterized by additional axioms. In the simplest case such an axiom is an Iswim definition. The resulting modification is called a “definitional extension” of the original system.

Iswim is an attempt at a general purpose system for describing things in terms of other things, that can be problem-oriented by appropriate choice of “primitives.” So it is not a language so much as a family of languages, of which each member is the result of choosing a set of primitives. The possibilities concerning this set and what
The commonplace expressions of arithmetic and algebra have a certain simplicity that most communications to computers lack. In particular, (a) each expression has a nesting subexpression structure, (b) each subexpression denotes something (usually a number, truth value or numerical function), (c) the thing an expression denotes, i.e., its “value”, depends only on the values of its subexpressions, not on other properties of them.

It is these properties, and crucially (c), that explains why such expressions are easier to construct and understand. Thus it is (c) that lies behind the evolutionary trend towards “bigger righthand sides” in place of strings of small, explicitly sequenced assignments and jumps.

The importance of denotational semantics was clearly highlighted.

\[ \psi(\nu_0 + \nu_1) = \psi(\nu_0) + \psi(\nu_1) \]

\[ \psi(\nu) = \text{Cond}(\psi(\nu_0), \psi(\nu_1); \psi(E)) \]

\[ \psi(\nu_0; E) = \psi(\nu_1; E) \]

Toward a Mathematical Semantics for Computer Languages

by

Dana Scott
Princeton University
and
Christopher Strachey
Oxford University

1971
Iswim brings into sharp relief some of the distinctions that the author thinks are intended by such adjectives as procedural, nonprocedural, algorithmic, heuristic, imperative, declarative, functional, descriptive. Here is a suggested classification, and one new word.

*denotative*

> **imperative**
The Next 700 Domain Specific Programming Languages
Datastructures and Recursion Schemes

Lists can be evaluated by folding their structure: cons and nil are replaced with a semantics of what to do

This technique generalises to many tree-like datatypes

There are many variations on this theme, depending on mutual recursion, recursion with parameters, recursion with historic context, and many more

\[
\text{foldr (+) 0 (2 : 7 : 1 : 8 : [])}
\]

\[
= 2 + 7 + 1 + 8 + 0 = 18
\]
Lists can be evaluated by folding their structure: cons and nil are replaced with a semantics of what to do
This technique generalises to many tree-like datatypes

There are many variations on this theme, depending on mutual recursion, recursion with parameters, recursion with historic context, and many more

Conjugate Hylomorphisms
On: The Mother of All Structured Recursion Schemes
Ralf Hinze
Nikita Wu
Jeremy Gibbons
Department of Computer Science, University of Oxford, Radcliffe Building, Parks Road, Oxford, OX1 3DL, England

Abstract
This paper studies how to extend a recursive study of structured recursion schemes. A general scheme in the form hyy = (asub, acat) produces a pair of subsequent recursion schemes, a sub-scheme that is fully applicative and a cat-scheme that is fully combinatorial, with two fundamental dualities.

Conjugate hylomorphisms arise from compositions in which both are fully applicative and both are fully combinatorial. The fundamental dualities combine into a single conjugate hylomorphism with two fundamental dualities.

Conjugate hylomorphisms arise from compositions in which both are fully applicative and both are fully combinatorial. The fundamental dualities combine into a single conjugate hylomorphism with two fundamental dualities.

\[ \text{foldr } (+) 0 (2 : 7 : 1 : 8 : []) = 2 + 7 + 1 + 8 + 0 = 18 \]
Domain-Specific Languages

```
data Expr where
  Add :: Expr → Expr → Expr
  Var :: String → Expr

data ExprF k where
  OpF :: k → k → k
  VarF :: String → k

alg :: ExprF a → a
alg (OpF a b) = a + b
alg (VarF x) = var x
```

2014

2014

```
[[x + y]] = [[x]] ⊕ [[y]]
```

```
eval alg (Add (Var x) (Var y)) = eval alg (Var x) + eval alg (Var y)
eval alg (Op m n) = alg (OpF (eval alg m) (eval alg n))
eval alg (Var x) = alg (VarF x)
```

```
la langue + de notations
```
Algebraic Effect Handlers

Syntax

Semantics

Syntax represented by the free monad for a functor that provides a signature

Semantics often in terms of a fold over the free monad

Abstract. We present an algebraic treatment of exception handlers and more generally introduce handlers for other computational effects representable by an algebraic theory. These include undetermination, interactive input/output, concurrency, alias, treat, and their combinations. In all cases the computation method is the free model of the theory. Each such handler corresponds to a model of the theory for the effects it holds. The handling construct, which applies a handler to a computation, is based on the one introduced by Plotkin and Ambriola and is interpreted using the homomorphism induced by the universal property of the free model. This general construct can be used to inverse proofs, mainly involving context from both theory and practice.

1 Introduction

In seminal work, Moggi proposed a uniform representation of computational effects by monads [32,55]. The computations that return values from a set $X$ are modeled by elements of $\mathbb{P} X$ for a suitable monad $\mathbb{P}$. Examples include exceptions, nondeterminism, interactive input/output, concurrency, state, time, combinations, and combinations thereof. Plotkin and Power later presented fold over the free monad methods for algebraic effects, that is, effects that allow a representation by equations and equations $p(x_1, \ldots, x_n) = x_0$.
Algebraic Effect Handlers

- Syntactic Programs are embedded syntax trees that can be inspected
Algebraic Effect Handlers

- **Syntactic Programs are embedded syntax trees that can be inspected**

- **Denotative**: The semantics is given by a handler that replaces operations with their meaning
Algebraic Effect Handlers

- **Algebraic**: Operations must respect substitution and sequencing
- **Syntactic**: Programs are embedded syntax trees that can be inspected
- **Denotative**: The semantics is given by a handler that replaces operations with their meaning
Algebraic Effect Handlers

- Algebraic: Operations must respect substitution and sequencing.
- Syntactic: Programs are embedded syntax trees that can be inspected.
- Extensible: Syntax trees can be extended with new syntactic nodes.
- Denotative: The semantics is given by a handler that replaces operations with their meaning.

\[
\begin{align*}
\text{do } x & \leftarrow \text{or } p \ q \quad = \quad \text{do } \text{or } (\text{do } x \leftarrow p \ ; \ k \ x) \\
\text{do } \text{fail} \quad & = \quad \text{do } \text{fail} \\
\text{do } \text{fail} \quad & = \quad \text{do } \text{fail} \\
\end{align*}
\]
Algebraic Effect Handlers

- Algebraic: Operations must respect substitution and sequencing
- Syntactic: Programs are embedded syntax trees that can be inspected
- Extensible: Syntax trees can be extended with new syntactic nodes
- Modulatative: The semantics is given by a handler that replaces operations with their meaning
- Modular: Handlers can be composed to give combinations of semantics

\[
\begin{align*}
do \ x & \leftarrow \ or \ p \ q \quad = \quad do \ or (do \ x \leftarrow \ p; \ k \ x) \\
& \quad (do \ x \leftarrow \ q; \ k \ x) \\
do \ fail & = \ do \ fail \\
k ()
\end{align*}
\]
Algebraic Effect Handlers

**Algebraic:** Operations must respect substitution and sequencing

**Syntactic:** Programs are embedded syntax trees that can be inspected

**Extensible:** Syntax trees can be extended with new syntactic nodes

**Flexible:** Multiple semantics can be given to a particular program

**Denotative:** The semantics is given by a handler that replaces operations with their meaning

**Modular:** Handlers can be composed to give combinations of semantics
Algebraic Effect Handlers

• Syntactic Programs are embedded syntax trees that can be inspected

• Equational: The relationship between operations can be expressed by laws

• Algebraic: Operations must respect substitution and sequencing

• Extensible: Syntax trees can be extended with new syntactic nodes

• Flexible: Multiple semantics can be given to a particular program

• Denotative: The semantics is given by a handler that replaces operations with their meaning

• Modular: Handlers can be composed to give combinations of semantics

\[
\begin{align*}
do x \leftarrow & \text{ or } p \; q & = & \text{ do } \text{ or } (\text{ do } x \leftarrow p; k \; x) \\
& & & (\text{ do } x \leftarrow q; k \; x) \\
do \text{ fail} & = & \text{ do fail} \\
do x \leftarrow & \text{ or } p \; q & = & \text{ do } x \leftarrow \text{ or } q \; p
\end{align*}
\]
Algebraic Effect Handlers

• Syntactic Programs are embedded syntax trees that can be inspected
• Extensible: Syntax trees can be extended with new syntactic nodes
• Modular: Handlers can be composed to give combinations of semantics
• Denotative: The semantics is given by a handler that replaces operations with their meaning

• Algebraic: Operations must respect substitution and sequencing
• Equational: The relationship between operations can be expressed by laws
• Pervasive: Algebraic effects cover a very large class of useful effects
• Flexible: Multiple semantics can be given to a particular program

\[
\text{do } x \leftarrow \text{ or } p \ q = \text{ do } \text{ or } (\text{do } x \leftarrow p; \ k \ x) \\
\quad \text{or } (\text{do } x \leftarrow q; \ k \ x)
\]

\[
\text{do } \text{fail} = \text{ do } \text{fail}
\]

\[
\text{do } x \leftarrow \text{ or } p \ q = \text{ do } x \leftarrow \text{ or } q \ p
\]

print get put throw fork read write

\[
\text{do } \text{fail} = \text{ do } \text{fail}
\]

\[
\text{do } x \leftarrow \text{ or } p \ q = \text{ do } x \leftarrow \text{ or } q \ p
\]
Algebraic Effect Handlers

- **Algebraic**: Operations must respect substitution and sequencing
- **Equational**: The relationship between operations can be expressed by laws
- **Pervasive**: Algebraic effects cover a very large class of useful effects
- **Syntactic**: Programs are embedded syntax trees that can be inspected
- **Extensible**: Syntax trees can be extended with new syntactic nodes
- **Flexible**: Multiple semantics can be given to a particular program
- **Modular**: Handlers can be composed to give combinations of semantics
- **Efficient**: Intermediate trees can be avoided, to immediately return results
- **Denotative**: The semantics is given by a handler that replaces operations with their meaning

\[
\text{do } x \leftarrow \text{or } p \ q = \text{do } \text{or} (\text{do } x \leftarrow p; k \ x) \\
\quad (\text{do } x \leftarrow q; k \ x)
\]

\[
\text{do } \text{fail} = \text{do } \text{fail} \\
\quad k ()
\]

\[
\text{do } x \leftarrow \text{or } p \ q = \text{do } x \leftarrow \text{or } q \ p
\]

- **print**
- **get**
- **put**
- **throw**
- **fork**
- **read**
- **write**
In a tree of nondeterministic computations, there are many different evaluation strategies.

For instance, breadth-first, depth-first, depth-bounded, single result etc.

This paper shows how these different strategies are handlers of nondeterminism.

This was used to model Prolog semantics as a DSL within Haskell.

Abstract

Heuristics and Handlers Combined

This paper shows how different evaluation strategies are handled in the context of nondeterministic computations. For instance, breadth-first, depth-first, depth-bounded, single result etc. This paper demonstrates how these strategies are handled in a tree structure.

Each semantics is given by a handler:

- **dfs**: [5, 3]
- **bfs**: [3, 5]
- **dbs 1**: [3]
- **once**: Just 5

This paper uses this approach to model Prolog semantics as a Domain-Specific Language (DSL) within Haskell.
Scoped Effects

list (return x) = [x]
list (fail) = []
list (or p q) = list p ++ list q

Once

once (return x) = Just x
once (fail) = Nothing
once (or p q) = case once p of
  Nothing -> once q
  Just x -> once p

Search

search (fail) = fail
search (return x) = return x
search (search p) = search p

Can the idea of algebraic operations be extended?

Many handlers have related behaviour

These are not algebraic operations
Theory Meets Practice

Algebraic effects are nice, but we can't express lots of constructs like if statements and try/catch as syntax.

Hmm, sounds like you need scoped effects ...

Wow! That works, but is it efficient?

Yes! It all fuses!

Amazing! 250x faster than our previous attempts! We've rolled this out to production!

Did you ever stop for a moment to think how many deep hours a feature like the new @github "jump to definition" saves a day?

It must be in the hundreds of hours across the planet per day!
Iswim brings into sharp relief some of the distinctions that the author thinks are intended by such adjectives as procedural, nonprocedural, algorithmic, heuristic, imperative, declarative, functional, descriptive. Here is a suggested classification, and one new word.

> imperative
References

- M. Davis, Engines of Logic, 2000
- M. Davis, The Universal Computer: The Road from Leibniz to Turing, Third Edition, 2018
- A. Hodges, Alan Turing: The Enigma, 2014
- D. Turner, Some History of Functional Programming Languages, 2012
- G. Leibniz, De Arte Combinatoria, 1666
- D. Hilbert, W. Ackermann, Grundzüge der theoretischen Logik, 1928
- J. Backus et al., The FORTRAN Automatic Coding System for the IBM 704, 1956
- F. Cajori, A History of Mathematical Notations, 1928
- K. Gödel, Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, 1931
- A. Church, A Set of Postulates for the Foundation of Logic, 1932
- A. Church, A Note on the Entscheidungsproblem, 1936
- A. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, 1937
- J. von Neumann, First Draft of a Report on the EDVAC, 1945
- J. McCarthy et al., LISP I Programmer's Manual, 1960
- J. Backus et al., Revised Report on the Algorithmic Language Algol 60, 1963
- P. Landin, The mechanical evaluation of expressions, 1964
- P. Landin, A correspondence between ALGOL 60 and Church's Lambda-notation: Part I, 1965
- D. Scott, C. Strachey, Towards a Mathematical Semantics for Computer Languages, 1971
- P. Landin, The Next 700 Programming Languages, 1966
- R. Hinze, N. Wu, J. Gibbons, Unifying Structured Recursion Schemes, 2013
- R. Hinze, N. Wu, J. Gibbons, Conjugate Hylomorphisms, 2015
- Z. Yang, N. Wu, Fantastic Morphisms and Where to Find Them, 2022
- G. Plotkin, M. Pretnar, Hanlders of Algebraic Effects, 2009
- M. Pirog, T. Schrijvers, N. Wu, M. Jaskelioff, Syntax and Semantics for Operations with Scopes, 2018
- P. Thomson, R. Rix, N. Wu, T. Schrijvers, Fusing Industry and Academia at GitHub, 2022

with thanks to Jeremy Gibbons and Jamie Willis for feedback on previous iterations of this talk